Reporte: Actividad 3

Alumna: Jocelyn Flores Gutiérrez

Maestro: Adalberto Emmanuel Rojas Perea

Materia: Estructura de Datos

Fecha: 07/09/2025

● Índice

---

1. ***Primer problema:*** *Serie de Fibonacci recursiva. Implementa una función recursiva en Java que calcule el enésimo número en la serie de Fibonacci. Asegúrate de incluir tanto el caso base como el caso recursivo.*
2. ***Segundo problema:*** *Suma de subconjuntos (Subset Sum). Desarrolla un algoritmo recursivo para determinar si existe un subconjunto de un conjunto dado de enteros que sume un valor objetivo.*
3. ***Tercer problema:*** *Algoritmo de backtracking para el problema del sudoku. Implementa un algoritmo de backtracking que resuelva un Sudoku. El programa debe llenar las celdas vacías del tablero de Sudoku dado.*

• Programas

Fibonacci sin recursividad:
En este caso se pretendió que el usuario indicara la posición del elemento que quería conocer de la serie Fibonacci y ese es el valor que se le pasa a la función, por lo que se utilizó un if y un ciclo for para abarcar todos los casos desde el principio.

```java
public static int fibonacciNormal(int n) {

    if (n == 1) {
        return 0;
    } else if (n == 2) {
        return 1;
    } else {
        int a = 0, b = 1, c = 0;
        for (int i = 1; i < n - 1; i++) { // En el primer caso (3) sólo se ejecuta 1 vez
            c = a + b;
            a = b;
            b = c;
        }
        return c;
    }
}
```

Se empezó a utilizar el ciclo for a partir del elemento 3 de la serie porque antes de ese el sistema no funciona correctamente. Se utilizaron las variables **a** y **b** para guardar los dos números previos y la variable **c** para guardar el valor actual.

El ciclo for se ejecuta el número de veces necesario para otorgarle el valor correspondiente a la variable **c**, cuyo valor es igual a la suma de los números anteriores guardados en las variables **a** y **b.** También actualiza los valores de estas últimas variables moviendo sus posiciones. Finalmente regresa el valor final obtenido de **c**.

Fibonacci con recursividad:
Espero poder explicarlo de la forma más entendible posible, porque además de lo difícil que es implementarlo, lo segundo más difícil es entender cómo funciona.

Se establecieron dos casos base principales: si el número es 0 o 1, regresa esos mismos valores, eso será importante porque la sucesión empieza con 0 y le sigue un 1, y a partir de ellos será que empezará a calcular los valores de la serie con la fórmula de Fn = Fn-1 + Fn-2. Básicamente se realizará la suma con los valores que regrese la misma función pero con los dos números anteriores, los cuáles dentro de la función realizarán el mismo procedimiento con los dos números anteriores hasta que lleguen a los casos base y vayan regresando el valor de su suma hasta llegar al primer caso (el valor que nos interesa) que hicimos.

```java
public static int fibonacciRecursivo(int n) {
    if (n == 0) {
        return 0; // Caso base
    } else if (n == 1) {
        return 1; // Caso base
    } else {
        return fibonacciRecursivo(n -1) + fibonacciRecursivo(n -2);
    }
}
```

Suma de subconjunto:
Estoy muy feliz de este código y espero haberlo implementado de forma considerablemente eficiente (muchos tutoriales de recursividad no son suficientes para tenerlo todo claro a la hora de implementar la recursividad, me pasé muchísimo tiempo pensando una solución), así que vamos a ello.

Primeramente en el Main se declararon los valores que se utilizarán en la función, como el arreglo de números enteros (el cual también se puede pedir al usuario), el tamaño del arreglo y un contador que será utilizado más adelante para recorrer los elementos del arreglo en sentido contrario a la variable del tamaño del arreglo.

```
int[] conjunto = {1,2,3,4,5,6,7,8,9}; // Conjunto de enteros pre-definido
int tamaño = conjunto.length, cont = 0;
```

En el Main, en las opciones del menú, se solicita al usuario el número al cuál se le quieren encontrar los sumandos y luego se utiliza un condicional para mostrar el mensaje correspondiente según el return de la función (si fue posible encontrar los sumandos o no).

```
case 2: // Suma de subconjunto
    System.out.print(s:"¿De qué número quiere obtener la suma? => ");
    int numero = input.nextInt();
    input.nextLine(); // Limpiar buffer
    if (subsetSum(conjunto, numero, tamaño -1, cont)) { // true false
        System.out.println("Sí es posible obtener " + numero);
    } else {
        System.out.println("No hay elementos que den la suma de " + numero);
    }
```

En la función se pretende que regrese un valor booleano (falso si no fue posible encontrar una combinación o si el arreglo se encuentra vacío), recibiendo como atributos las variables que le enviamos al llamarla.

Nota: Es importante tomar en cuenta que al llamar la función, en el tamaño, se le envía el tamaño total del arreglo -1.

La primera sentencia, que funciona también como caso base, compara si el arreglo está vacío o si ya recorrió todo y no se encontraron sumandos válidos. Si no es el caso, entonces se procede consultando si el valor del índice más grande del arreglo es mayor o igual que el número proporcionado por el usuario (porque si lo fuera, no sería posible que su suma con otro número fuera igual al número que nos dieron).

Lo que se pretende con la función es recorrer el arreglo desde el último elemento hasta el índice 0 con la variable tamaño, y al revés con la variable cont, siendo cuidadosos de no comparar el mismo número.

Si el número que está en el índice más grande resulta igual o mayor a la suma, se ignora llamando de nuevo a la función pero pasando al próximo índice más grande (dejando la variable *cont* igual porque tiene valor de 0 siempre que llega a esa parte).

Si no se cumple la anterior condición entonces se verifica si al recorrer el arreglo en los dos sentidos se llega al mismo número y se evita comparar el mismo valor, entonces se llama de nuevo a la función pero esta vez le resta 1 al tamaño máximo en el que va.

La siguiente condición pretende comparar finalmente si los elementos en las posiciones actuales tienen una suma igual a la que se busca, y si es el caso entonces procede a imprimir los números que lo lograron y regresar *true* como valor.

Si aún no se cumple esa condición, y los dos números en las posiciones actuales no dan la suma que se busca, entonces procede a llamarse la función aumentando el contador para que empiece a recorrer el arreglo desde las primeras posiciones hasta las últimas.

```java
public static boolean subsetSum(int[] conjunto, int numero, int tamaño, int cont) {
    if (tamaño == 0) { // Si está vacío el conjunto o ya recorrió todo y no encontró nada
        return false;  // Caso base
    } else {
        if (conjunto[tamaño] >= numero) { // Si el elemento es mayor o igual al número se ignora
            return subsetSum(conjunto, numero, tamaño - 1, cont);
        } else {
            if (cont == tamaño) {
                return subsetSum(conjunto, numero, tamaño -1, cont = 0);
            } else {
                if (conjunto[cont] + conjunto[tamaño] == numero) {
                    System.out.print(" "+ conjunto[cont] + " + " + conjunto[tamaño] + " ");
                    return true;
                } else {
                    return subsetSum(conjunto, numero, tamaño, ++cont);
                }
            }
        }
    }
}
```

Nota: Está cortito y bonito el código :')

• Código JAVA

- Main -

```
import java.util.Scanner;
public class Main {

    public static int fibonacciNormal(int n) {
        if (n == 1) {
            return 0;
        } else if (n == 2) {
            return 1;
        } else {
            int a = 0, b = 1, c = 0;
            for (int i = 1; i < n - 1; i++) { // En el primer caso (3) sólo se ejecuta 1
vez
                 c = a + b;
                 a = b;
                 b = c;
            }
            return c;
        }
    }
    public static int fibonacciRecursivo(int n) {
        if (n == 0) {
            return 0; // Caso base
        } else if (n == 1) {
            return 1; // Caso base
        } else {
            return fibonacciRecursivo(n -1) + fibonacciRecursivo(n -2); // Llamadas a sí
mismo
        }
    }
    public static boolean subsetSum(int[] conjunto, int numero, int tamaño, int cont) {
        if (tamaño == 0) { // Si está vacío el conjunto o ya recorrió todo y no encontró
nada
            return false;  // Caso base
        } else {
            if (conjunto[tamaño] >= numero) { // Si el elemento es mayor o igual al número
se ignora
                return subsetSum(conjunto, numero, tamaño - 1, cont);
            } else {
                if (cont == tamaño) {
                    return subsetSum(conjunto, numero, tamaño -1, cont = 0);
                } else {
                    if (conjunto[cont] + conjunto[tamaño] == numero) {
                        System.out.print(" "+ conjunto[cont] + " + " + conjunto[tamaño] + "
");

                        return true;
                    } else {
                        return subsetSum(conjunto, numero, tamaño, ++cont);
                    }
                }
            }
        }
    }
```

```
    //public static void solucionSudoku(int[] matriz) {}

    public static void main(String[] args) {

        // Primer problema: Serie de Fibonacci recursiva. Implementa una función recursiva
en Java que calcule el enésimo número en la serie de Fibonacci. Asegúrate de incluir tanto
el caso base como el caso recursivo.
        // Segundo problema: Suma de subconjuntos (Subset Sum). Desarrolla un algoritmo
recursivo para determinar si existe un subconjunto de un conjunto dado de enteros que sume
un valor objetivo.
        // Tercer problema: Algoritmo de backtracking para el problema del sudoku.
Implementa un algoritmo de backtracking que resuelva un Sudoku. El programa debe llenar las
celdas vacías del tablero de Sudoku dado.
        Scanner input = new Scanner(System.in);
        boolean ciclo = true;
        int[] conjunto = {1,2,3,4,5,6,7,8,9}; // Conjunto de enteros pre-definido
        int tamaño = conjunto.length, cont = 0;

        while(ciclo) {
            System.out.println("\n ==== Menú de acciones ====");
            System.out.print("1) Serie fibonacci \n" +
                             "2) Suma de subconjunto \n" +
                             "3) Sudoku \n" +
                             "4) Salir \n" +
                             "=> ");
            int seleccion = input.nextInt();
            input.nextLine(); // Limpiar buffer
            switch(seleccion) {
                case 1: // Serie fibonacci
                    System.out.print("¿Qué elemento quiere ver de la serie fibonacci? =>
");
                    int topeFibonacci = input.nextInt();
                    input.nextLine(); // Limpiar buffer
                    System.out.println("Fibonacci normal: " +
fibonacciNormal(topeFibonacci));
                    System.out.println("Fibonacci con recursividad: " +
fibonacciRecursivo(topeFibonacci));
                    break;
                case 2: // Suma de subconjunto
                    System.out.print("¿De qué número quiere obtener la suma? => ");
                    int numero = input.nextInt();
                    input.nextLine(); // Limpiar buffer
                    if (subsetSum(conjunto, numero, tamaño -1, cont)) { // true false
                        System.out.println("Sí es posible obtener " + numero);
                    } else {
                        System.out.println("No hay elementos que den la suma de " +
numero);
                    }
                    break;
                case 3: // Sudoku
                    System.out.println("Llamada a la función del sudoku. ");
                    // sudokuSolucion(int[] matriz);
                    break;
                case 4: // Salir
                    ciclo = false;
                    break;
                default:
```

```
                    System.out.println("Opción inválida.");
                }
            }
        }
}
```

● Casos de uso

Menú principal:

```
 ==== Menú de acciones ====
1) Serie fibonacci
2) Suma de subconjuntos
3) Sudoku
4) Salir
=>
```

Fibonacci:
(Sin recursividad: empieza a contar desde 0, ejemplo: 0, 1, 1, 2, 3, 5, 8, 13.
Con recursividad: empieza a contar desde 1, ejemplo: 1, 1, 2, 3, 5, 8, 13)

```
 ==== Menú de acciones ====
1) Serie fibonacci
2) Suma de subconjuntos
3) Sudoku
4) Salir
=> 1
¿Qué elemento quiere ver de la serie fibonacci? => 1
Fibonacci normal: 0
Fibonacci con recursividad: 1
```

```
 ==== Menú de acciones ====
1) Serie fibonacci
2) Suma de subconjuntos
3) Sudoku
4) Salir
=> 1
¿Qué elemento quiere ver de la serie fibonacci? => 5
Fibonacci normal: 3
Fibonacci con recursividad: 5
```

Suma de subconjunto:

```
int[] conjunto = {1,2,3,4,5,6,7,8,9}; // Conjunto de enteros pre-definido
```

```
 ==== Menú de acciones ====
1) Serie fibonacci
2) Suma de subconjunto
3) Sudoku
4) Salir
=> 2
¿De qué número quiere obtener la suma? => 5
 1 + 4 Sí es posible obtener 5
```

```
 ==== Menú de acciones ====
1) Serie fibonacci
2) Suma de subconjunto
3) Sudoku
4) Salir
=> 2
¿De qué número quiere obtener la suma? => 9
 1 + 8 Sí es posible obtener 9
```

```
 ==== Menú de acciones ====
1) Serie fibonacci
2) Suma de subconjunto
3) Sudoku
4) Salir
=> 2
¿De qué número quiere obtener la suma? => 0
No hay elementos que den la suma de 0
```

```
 ==== Menú de acciones ====
1) Serie fibonacci
2) Suma de subconjunto
3) Sudoku
4) Salir
=> 2
¿De qué número quiere obtener la suma? => 1
No hay elementos que den la suma de 1
```

● Conclusión

La lógica de programación siempre debe procurar resolver los problemas de la forma más eficiente, breve y efectiva para ahorrar los recursos del sistema y la memoria; y, a veces, una forma de lograrlo es implementando recursividad y algoritmos de divide y vencerás.

La recursividad permite ahorrar, simplificar el tamaño del código y reutilizarlo para solucionar el problema llamando a la función dentro de la misma función o a través de otra (recursividad directa e indirecta). Si bien puede ser muy eficiente, a veces también no es la mejor opción para resolver un problema sencillo porque consume mucho tiempo de ejecución, puede resultar más tardado pensar en una adaptación que utilice recursividad y así mismo, toma más tiempo comprender el funcionamiento de un código con recursividad.

El mismo principio de la recursividad se encuentra en la metodología de “divide y vencerás”, que en muchas ocasiones utiliza la recursividad para dividir un problema grande en partes más pequeñas y sencillas de resolver. El principio de “divide y vencerás” es bastante eficiente para resolver problemas grandes que requieren de un mismo procedimiento pero a distinta escala, siguiendo por lo general los pasos de “dividir”, “conquistar” y luego “combinar”, justamente como lo hace la recursividad.

Por si fuera poco, el backtracking también involucra la recursividad para analizar múltiples decisiones y decantarse por la mejor alternativa, o para probar cuántas posibles soluciones hay: es una especie de prueba y error, donde si uno se equivoca sólo debe volver atrás. Según pude ver en los ejemplos, el backtracking toma la forma de un árbol, donde va recorriendo cada rama hasta encontrar la mejor opción y entonces ya no tiene que recorrer el resto de ramas. Se suele utilizar para casos como el sudoku, los laberintos o cuando quieren conocer todas las posibles soluciones de algo.

Todos estos algoritmos son herramientas que, como todo, si encuentras el caso donde más pueden funcionar entonces resultarán en un método eficiente, sencillo y fácil de aplicar, pero fuera de eso, a veces no son la mejor opción.

Mi experiencia con esta actividad es que la teoría es, de cierta manera, sencilla de entender. El concepto de recursividad es una función que se llama a sí misma dentro de la misma función; la metodología de divide y vencerás es también prácticamente la recursividad, y el backtracking consiste en recorrer las diferentes opciones a prueba y error, y en caso de equivocarse, retroceder hasta el último punto seguro (también implementando recursividad). Pero independientemente de lo sencillo que resulte entender la teoría, llevarlo a cabo en código (y adecuarlo al problema que tengas) es algo más difícil de lo que parece y que supone un gran reto.
Realmente pude observar que se logran resultados muy interesantes y exitosos con la recursividad, sin embargo requiere de mucha práctica para dominarlo. Al final no pude llevar a cabo el tercer caso del sudoku con backtracking, pero definitivamente voy a seguir practicando con el backtracking porque en esta ocasión superó mi entendimiento.